Available online at www.sciencedirect.com
ScienceDirect
Pervasive and Mobile Computing 3 (2007) 413–438
pervasive and mobile computing
www.elsevier.com/locate/pmc

An application adaptation layer for wireless sensor networks☆

M. Avvenuti, P. Corsini, P. Masci, A. Vecchio*

Dipartimento di Ingegneria dell'Informazione, Università di Pisa, Via Diotisalvi 2, I-56122 Pisa, Italy

Received 31 October 2006; received in revised form 15 April 2007; accepted 16 April 2007
Available online 20 April 2007

Abstract

In wireless sensor networks, poor performance or unexpected behavior may be experienced for several reasons, such as trivial deterioration of sensing hardware, unsatisfactory implementation of application logic, or mutated network conditions. This leads to the necessity of changing the application behavior after the network has been deployed. Such flexibility is still an open issue as it can be achieved either at the expense of significant energy consumption or through software complexity. This paper describes an approach to adapt the behavior of running applications by intercepting the calls made to the operating system services and changing their effects at run-time. Customization is obtained through small fragments of interpreted bytecode, called *adaptlets*, injected into the network by the base station. Differently from other approaches, where the entire application is interpreted, adaptlets are tied only to specific services, while the bulk of the application is still written in native code. This makes our system able to preserve the compactness and efficiency of native code and to have little impact on the overall application performance. Also, applications must not be rewritten because the operating system interfaces are unaffected. The adaptation layer has been implemented in the context of TinyOS using an instruction set inspired to the Java bytecode. Examples that illustrate the programming of the adaptation layer are presented together with their experimental validation.

Keywords: Wireless sensor network; Middleware; Programming support; Adaptation of applications

☆ This work was partially supported by Fondazione Cassa di Risparmio di Pisa, Italy (SensorNet Project).
* Corresponding author.
E-mail addresses: marco.avvenuti@iet.unipi.it (M. Avvenuti), paolo.corsini@iet.unipi.it (P. Corsini), paolo.masci@iet.unipi.it (P. Masci), alessio.vecchio@iet.unipi.it, a.vecchio@ing.unipi.it (A. Vecchio).

1574-1192/$ - see front matter
doi:10.1016/j.pmcj.2007.04.001

1. Introduction

Wireless sensor networks (WSNs) are massively distributed systems where low cost and low power nodes are used to monitor a physical phenomenon. WSNs do not require any external infrastructure and can scale to hundreds or thousands of nodes [1]. Since they provide fine-grain monitoring over a large time scale, sensor network applications can play a key role in many different areas [2]. Examples include intrusion detection and surveillance [3], wildlife monitoring [4], precision agriculture [5], structural monitoring [6], and sniper detection [7]. The physical proximity of sensors to the observed phenomenon and the programmability of nodes can increase the level of accuracy and adaptability of the sensing process with respect to traditional solutions.

Computing and communication capabilities of individual nodes are usually very limited. Such constraints pushed the development of operating systems specifically designed to fit the available resources. Moreover, operations in a WSN are fundamentally constrained by energy availability: nodes are battery operated, and periodic recharging is usually not possible. Thus, energy management is a major concern at every software layer.

Applications for WSNs are typically long-lived, hence, in many cases, it might be necessary to modify the behavior of executed applications [9]. The need for application updates may originate from a variety of reasons. Frequently, bugs or unsatisfactory implementation become manifest only after the software has been deployed. This is particularly true in an error-prone environment such as WSNs where, besides the inherent complexities bound to the development of software in a distributed environment, implementation is made more difficult by the absence of high level communication and coordination abstractions. Further, a priori assumptions about the operating environment may not hold well in practice, or the sensing equipment may return readings different than those expected because of the manufacturing process [10]. All these cases may represent a source of malfunctions or lead to poor performance, since applications are highly sensible to the values gathered from the environment.

The size of software updates may range from a parameter change to a complete reprogramming of the software installed on nodes. Existing tools used to manage software updates in PC-class devices cannot be adopted in the WSN domain because of the limited resources available on sensor nodes. This has stimulated the definition of systems for the management of software updates that are inexpensive in terms of energy requirements and hardware resources.

1.1. Existing approaches to reprogramming

Applications can be executed on a sensor node as native code or inside a virtual machine. The execution model determines the strategies that can be adopted for application reprogramming. Here, we briefly recall the main features and limits of available solutions. Further information can be found in [11] and in Section 6.

1.1.1. Native code

Native code is generated by a cross compiler that translates source code written in a high level language into instructions suitable for the type of micro-controller available on

the sensor nodes. The result of the compilation is a single binary image that contains the application logic together with all necessary libraries and components of the operating system (e.g., scheduler and drivers). Then, the image is copied into each sensor node by means of a wired connection. Native code is executed by the micro-controller without additional overhead. For this reason, native code is currently used to implement almost all applications for WSNs.

When applications are executed as native code, the possible reprogramming strategies are full image replacement, incremental reprogramming, and loadable modules.

Full image replacement is the simplest way of reprogramming nodes in a sensor network: a completely new binary image is created and then transferred to the nodes via a wireless connection [12,13]. The new image is first stored in the external flash memory, since the running image cannot be replaced on-the-fly. Then, the new image is loaded into the program memory of the micro-controller, and the execution of the new image takes place after a reboot. The main advantage of this solution is its simplicity. However, full image replacement presents two negative aspects: first, it is very expensive in terms of energy consumption (the whole image has to be transferred and replaced even if a limited amount of changes is required); second, continuity of service is prevented and the execution state is lost since a reboot is always needed.

Incremental reprogramming reduces the energy consumption required by the previous technique by sending only the incremental changes for the new program version [14, 15]. The procedure comprises the following steps. The base station determines and distributes the differences between the two program images. Each node involved in the reprogramming process copies its current program image from the program memory to the external flash memory, and applies the changes using the code received from the network. Then, the patched image is moved back to the program memory and is executed after a reboot. This solution reduces the size of transferred data and is particularly effective when the new program image is similar to the old one (e.g., in case of correction of small software bugs, and changing of parameters). However, the obtained gain is limited by two factors: (i) small changes in the source code can produce large differences in the binary forms, (ii) the construction of the patched image requires several accesses to the external flash memory (to reduce energy consumption, the external memory should be used only when it is necessary [16,17]).

With loadable modules, the application is made up of several parts that can be loaded and replaced at run-time [18]. Since modules can be replaced independently, this approach limits the amount of code that must be transferred during application reprogramming. Dynamic loading of modules requires support from the operating system, since new modules must be linked together with other modules (references to functions defined externally must be resolved to their real physical addresses before execution). The communication cost between modules is generally higher than the single image approach, because of an additional level of indirection.

1.1.2. Virtual machines

Virtual machines create a run-time environment that isolates the execution of applications from the underlying platform [19,8]. The execution engine (interpreter) is provided with a high level instruction set, and this enables a compact representation of the

application code. Thus, with respect to native code, the use of a virtual machine reduces the size of applications and ease the distribution of software updates. Programs can be stored in RAM, and additional energy is not required to store and retrieve code from the external flash memory. However, since execution takes place within a software machine, execution cost is higher if compared to native code.

1.2. Our contribution

We propose a hybrid approach for reconfiguration of applications that combines the efficiency of execution of native code with the flexibility and ease of programming of virtual machines. The main thrust is to provide a practical and effective mechanism to configure and tune applications for WSNs, and not to implement an alternative solution to the problem of reprogramming nodes from scratch.

Applications can be adapted to mutated operating conditions through the intervention of an adaptation layer that is seamlessly inserted at the interface between the application and the operating system. The adaptation layer intercepts calls to primitives and library functions made by the application, and modifies the effects of such calls by executing specific fragments of code, called *adaptlets*.

The adaptation layer is distinct from the application, and embeds a tiny virtual machine that executes adaptlets. As the adaptation layer is not visible at the application level, applications can still be developed according to the "traditional" way: they are written in a high level language on the basis of the standard APIs offered by the operating system, then they are compiled into machine instructions without any awareness of the adaptation layer. This guarantees the re-usability of existing applications with minimal changes.

Adaptlets are written in a mid-level language that is similar to the Java bytecode. New adaptlets, or new versions of already installed ones, are spread over the network by the base station. Sensor nodes associate them to relevant operating system services. Adaptlets are small: they can be stored and executed directly in RAM, and their distribution and execution introduces little overhead. Furthermore, since the bulk of the application runs as native code, the overall performance can be preserved.

As demonstrated by the examples given in Section 4, the adaptation layer successfully supports different forms of reconfiguration such as threshold tuning, filtering noise, and in-network data processing.

2. The application adaptation layer

The adaptation layer works by intercepting the calls made by the application to the operating system services. It should be noted that, although spread over a broad spectrum of usages, software running on sensor nodes is characterized by some recurring patterns: applications typically require to collect data from the environment and to communicate with a base station (usually through the cooperation of other nodes). This suggested us to consider for adaptation purposes only the calls made to the data acquisition library and the communication library (Fig. 1).

Other characteristics of the adaptation layer are the following:

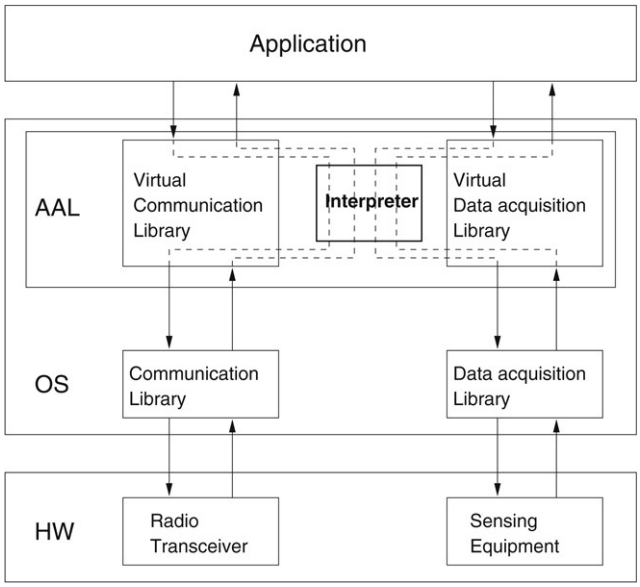


Fig. 1. The application adaptation layer (AAL).

- different adaptlets can be associated to different operating system services;
- executed adaptlets are dynamically changed from the base station through mechanisms that disseminate the code into the network;
- the size of the adaptation layer is in the order of few tens of kilobytes;
- the instruction set is easily configurable at compile-time and allows developers to generate compact adaptlets.

Virtualizing the data acquisition library. A way to modify the behavior of an application is to change its "perception" of the environment, rather that changing its code [24,25]. In particular, a layer of computation can be used to expose the application to new sensed values. This component, that we call *virtual data acquisition library* (VAL), is placed between the application and the sensor acquisition library provided by the operating system. Whenever the application senses the environment, VAL intercepts the call, and an adaptlet is executed. While the standard library returns raw data to the application, VAL can (i) read and process the locally sampled values, e.g. to filter spikes or calibrate the sensing hardware, (ii) request the sampled values from neighbouring nodes, e.g. to aggregate the values of the surrounding area, (iii) perform computation to feed the application with artificially generated values, e.g. to reproduce operating conditions before real deployment.

Virtualizing the communication library. The adaptation layer can modify the communication pattern of the application by virtualizing the communication library. The *virtual communication library* (VCL) is placed between the application and the communication library, and is capable of running different adaptlets for outgoing and ingoing packets. Every time the application sends out a packet, VCL intercepts the call and an adaptlet is executed. The adaptlet can modify the packet generated at the application level, e.g. to change the destination address, or to compress/cipher the application payload.

Then, the modified packet can be transmitted through the wireless interface. Similarly, VCL intercepts all packets that are received via the radio transceiver. On packet arrival, a specific adaptlet is executed. This program can modify the received packet before delivering it to the application, for example to decompress/decipher the payload.

2.1. The instruction set

The instruction set is derived from a subset of the Java bytecode (the mnemonic form is listed in Appendix). The instruction set includes general purpose instructions, used to access registers and stack and to perform arithmetic and branching operations. Operands can be immediate, contained in registers, or placed on top of the stack. Besides general purpose instructions, the interpreter also includes some instructions specifically designed for sensor network applications. For example, to compute frequent high level operations, to get data from the local sensor board, to request data sensed by neighbouring nodes, and to send a packet.

Registers and Stack. Instructions that operate on registers and stack are used to store and retrieve temporary values and to fetch operands. Examples of instructions are `LOAD r`, that copies the value of register `r` onto the stack, and `POP`, that removes the topmost value from the stack.

Branching. Branching instructions enable the execution of different instruction sequences depending on run-time conditions. An example of a branching instruction is `GOTO l`, that jumps to instruction number *l*.

Arithmetic. Besides basic arithmetic operations, such as sum and subtraction, the interpreter includes some instructions to compute frequent high level operations, as for example the average (`AVG`), the minimum (`MIN`) and the maximum (`MAX`) values.

Data acquisition. These instructions are specialized for getting data from sensors. For example, `ADCGET` gets the ADC reading from the real sensor, `ADCREQ` requests the ADC reading from the neighbouring nodes.

Network. Some special instructions can be used to send messages through the wireless interface (`SEND`), or to notify the application that a message has been received (`RECEIVE`).

3. System architecture and implementation

The adaptation layer is presented within the context of the TinyOS programming model. TinyOS [26] is an open-source operating system designed for wireless sensor networks. It is a popular platform and it is available for a number of different hardware architectures.

3.1. TinyOS basic concepts

Applications that run on TinyOS, as well as TinyOS itself, are written in nesC [27], a component-oriented extension of the C programming language. With nesC, programmers can define new components using a C-like syntax, and they can connect together existing components to create new components or applications (the act of connecting components

together is called "wiring"). Each component declares input and output functions, called *events* and *commands*, that are used in the wiring process. Commands and events are usually grouped into *interfaces*, i.e. labelled sets with a given type. The binary form of the application installed on a node includes the operating system, the used services and the user-defined application logic. The binary file is usually uploaded to nodes before they are deployed on the field.

In TinyOS, the data acquisition library provides analog to digital converter functions and is implemented by software components declaring the ADC interface. The ADC interface includes a command and an event[1]: the *getData()* command is non-blocking, and is called by the application to initiate a reading. The *dataReady(...)* event is signalled by the sensor–driver when the acquisition process terminates. The event function has one parameter, which is the value of the sensor reading.

The communication library allows the application to send and receive messages through the parametric interfaces *SendMsg[uint8_t id]* and *ReceiveMsg[uint8_t id]*.[2] These parametric interfaces are used to send/receive *active message*s [28], which are messages characterized by an integer identifier. The identifier is used to associate messages to software channels. In other words, messages sent via *SendMsg[i]* are received through *ReceiveMsg[i]*, and a specific handler is executed.

The *SendMsg* interface includes a command and an event. The *send(...)* command has three parameters: the destination address, the length (in bytes) of the payload, and a pointer to the buffer containing the message to be sent. The command is non-blocking. The *sendDone(...)* event is signalled when the message is actually transmitted. The event has two parameters: a pointer to the buffer containing the last message sent, and a flag reporting the success or failure of the sending attempt.

The *ReceiveMsg* interface defines only the *receive(...)* event, which is signalled every time a message is successfully received by the node. The event has one parameter, that is a pointer to the buffer containing the received message. The event must return the buffer for the next receive operation to the caller.

All components provide the *SplitControl* interface that can be used by the application to initialize, turn on/off the associated hardware components.

3.2. Interfaces and functionalities of the adaptation layer

The adaptation layer is a component that must be wired to the main application, on one side, and to the real network and sensor components provided by the operating system, on the other side (Fig. 2). The layer encapsulates a stack-based virtual machine that interprets the bytecode described in Section 2.1.

Within the adaptation layer, VAL is implemented by a component that provides the interfaces ADC and *SplitControl*, the same interfaces exported by the TinyOS sensor–driver components. Each time the application calls the *getData()* command, VAL executes the associated adaptlet. On adaptlet completion, the final value is left on the stack

[1] The ADC interface also includes the *getContinuousData()* command, used by applications to initiate a series of ADC conversions. For the sake of clarity, this command is not considered in our description.

[2] This notation means that 256 instances of SendMsg and ReceiveMsg are available.

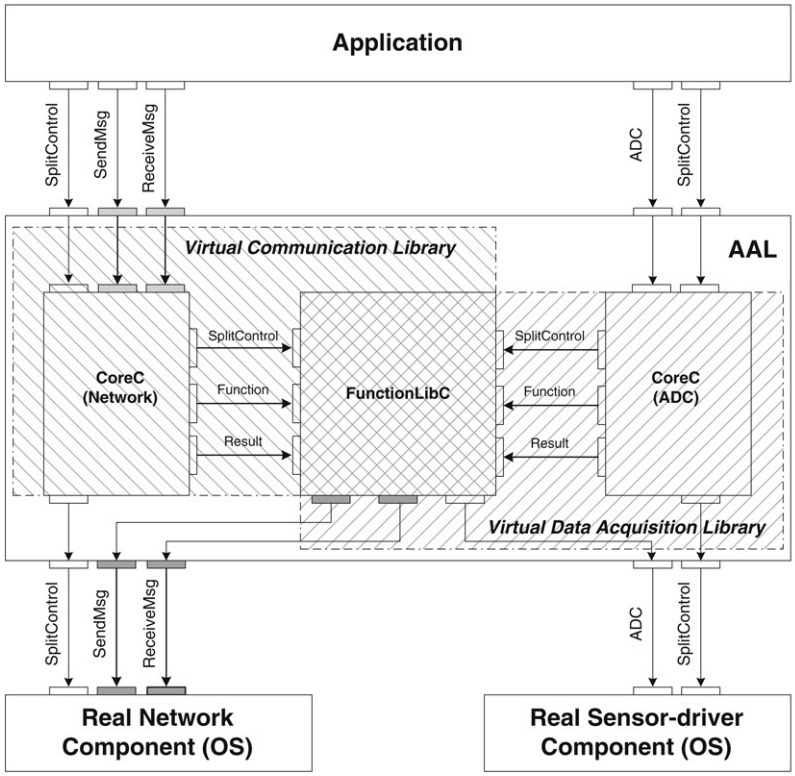


Fig. 2. CoreC components intercept the calls issued by the application, and control the execution flow of adaptlets. FunctionLibC implements the instruction set. Function and Result interfaces are used by CoreC components to trigger the execution of instructions and to retrieve results. FunctionLibC uses the SendMsg and ReceiveMsg interfaces to obtain network connectivity.

top by the adaptlet, and is returned to the application by signalling the *dataReady(...)* event. In turn, VAL uses the interfaces *SplitControl* and ADC provided by the sensor–driver component to interact with the real sensing equipment, when needed.

Similarly, VCL provides the parametric interfaces *SendMsg* and *ReceiveMsg*, the same interfaces provided by the TinyOS network component. Each time the application calls the *send(...)* command (of the *SendMsg* interface), an adaptlet is executed. On adaptlet completion the application is notified through the *sendDone(...)* event. The adaptlet leaves the arguments of the *sendDone(...)* event on the stack top in order to use them as actual parameters when notifying the event to the application. Likewise, each time a packet is received by the node, another adaptlet is executed. When the adaptlet completes, the *receive(...)* event (of the *ReceiveMsg* interface) is signalled to the application. Also in this case, the values that shall be returned to the application, i.e. the arguments of the *receive(...)* event, are left on the stack by the adaptlet.

The interpretation is carried out by two different kinds of components: *CoreC*, that intercepts the calls to the operating system services and controls the execution flow of the

running adaptlet, and *FunctionLibC*, that contains the implementation of the instruction set. *CoreC* fetches the current instruction and its operands, if any, then dispatches the instruction opcode to *FunctionLibC* through a command of the *Function* interface. *FunctionLibC* executes the instruction and returns a value to *CoreC*, which in turn uses this value to update the program counter. This process is repeated until the end of the adaptlet is reached. On adaptlet termination, *CoreC* retrieves the final value of the computation through the *Result* interface, and notifies the application through either the *dataReady(...)*, or the *sendDone(...)* or the *receive(...)* event. As shown in Fig. 2, the layer includes two instances of *CoreC*: one is used to execute adaptlets when the application accesses the communication library, the other is used when the application accesses the data acquisition library. This way, different adaptlets can be executed concurrently by the interpreter.

Inside *FunctionLibC* each instruction is implemented by a separate component. Components implementing the instructions are interchangeable at compile-time, thus the set of implemented instructions can be easily extended by adding programmer-defined functional components [29]. Instructions are executed within lightweight tasks [27], thus the original responsiveness of the main application is preserved. *FunctionLibC* is stateless and can be shared by *CoreC* modules.

By means of the connection with the real communication library, the adaptation layer can (i) receive packets containing adaptlets, (ii) forward adaptlets to nodes that are not within the communication range of the base station, (iii) communicate with the adaptation layer of neighbouring nodes (e.g., to request their sampled values or to reply to a request).

3.3. Dissemination of adaptlets

Adaptlets are usually few bytes long. As a consequence, they can be contained within a single packet, and can be efficiently disseminated over the network even using a simple communication protocol. Moreover, adaptlets can be stored directly in RAM, thus saving energy with respect to other approaches that require read/write operations on the flash memory. When a new adaptlet is received, it is associated to a specific function of the operating system on the basis of an identifier contained within the packet.

The default protocol for adaptlets dissemination is based on broadcast: every node retransmits the packet once, until all the nodes have been reached. It is known that such a flooding scheme can be adequate when packets must be delivered to almost all nodes in the network, and the network is reasonably sparse. However, flooding is not efficient when the network is particularly dense, or if packets must be delivered to a small fraction of the nodes. We did not further deal with this problem, which is orthogonal to the one of dynamic application reconfiguration and has been extensively studied. For example, intelligent flooding [30] can be a good solution in case of dense networks, while geocast-based protocols can provide an efficient way to disseminate information in geographically limited regions [31]. The reader is also directed to [20–23] for additional information.

3.4. Generating code for adaptlets

Generating code directly at the bytecode level can be cumbersome. Thus, we used the standard Java compiler to ease the production of adaptlets. After compilation, the *class*

file has to be processed by a post-compiler tool that produces the final adaptlet code. We implemented this tool by using the ByteCode Engineering Library (BCEL) [32], an open-source library that provides an object-oriented view of binary class files. Basically, we used this library to parse the binary file generated by the compiler, and then to translate the Java bytecode to instructions suitable for adaptlets. In order to use this tool, the programmer has to define a class composed as follows:

- Fields $r_0, \ldots, r_n$ (*int* type): they represent the registers of the interpreter, and they maintain their values across different executions of the adaptlet. The first time the adaptlet is executed, the registers contain the default value 0.
- Fields $a_0, \ldots, a_n$ (*byte[]* type): they represent buffers that store packets. They are used only by the CoreC/Network. By default, every time the execution of an adaptlet starts, register a_0 contains a copy of the packet passed as parameter to the *send(...)* command or the *receive(...)* event.
- Field *LOCAL_ADDRESS*: it represents the actual node id, and can be used to customize the execution on the basis of the node address.
- Methods that represent the special instructions of the interpreter, such as `AVG` and `SEND`, must be declared. The return type and the arguments of such methods shall be compliant with the corresponding instructions.
- Method *init()*: it contains the initialization code for the interpreter and it is executed only when the adaptlet is installed.
- Method *execute()*: it implements the actual behavior of the adaptlet. The method can declare local variables of type *int* to store temporary results, use the class fields r_i and a_i, and invoke class methods.

Examples of Java classes that can be translated into adaptlets are given in Section 4.

4. Demonstrative adaptlets and experimental validation

In the following, we show some uses of the adaptation layer and give a simple experimental validation of the proposed solutions. All experiments were carried out by using Tmote [33] nodes, based on the Telos-B architecture. For the simplest examples we show the bytecode of the adaptlet (see Appendix for the bytecode specification), while in the other cases we give the Java source code used to generate the adaptlet code.

4.1. Removing spikes

If the physical phenomenon under observation is characterized by large and frequent fluctuations, the acquisition of a single sample from the environment can be scarcely significant. In such a situation, a more meaningful sampling can be achieved, without changing the code of the application, by instructing VAL to act as a low-pass filter that seamlessly filters and aggregates samples. For example, the following adaptlet removes fluctuations by sampling the ADC once a second and averaging four samples:

```
0:PUSH 10     (push the sampling period, expressed in 100's of
               milliseconds, onto the stack)
```

```
1:PUSH 4        (push the number of requested samples onto the stack)
2:ADCGET        (pop two operands from the stack, periodically get
                the ADC samples as specified by the operands, and
                place the sampled values onto the stack. An integer
                specifying the number of actual sampled readings is
                placed on the stack top)
3:AVG           (compute the average of the values on the stack)
4:RETURN        (program end)
```

We experimentally validated such solution with a sensor network used to monitor the level of light inside a building (e.g., if some lights are turned on after the closing time, then they must be switched off). A light is considered as turned on if the observed level of light is greater than a given threshold, otherwise the light is considered off. Fig. 3(a) shows the values of samples when a light is turned off (on the left-hand side of the graph) and when it is turned on (on the right-hand side).

Based on this knowledge, we developed the application and set the value of the threshold to T. The threshold is slightly higher than the mean value, in order to compensate for possible light coming from outside. After the deployment of the network, the values acquired by some nodes were lower than T even if the light was turned on. The malfunction was due to the fact that a number of incandescence lamps were replaced with fluorescent lamps.

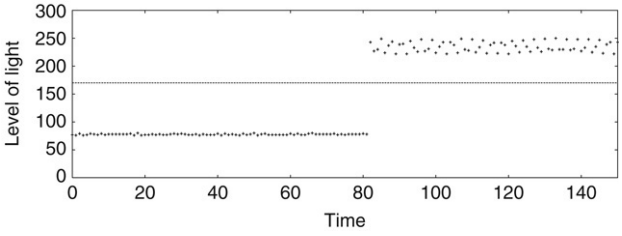
Fig. 3(b) shows the sampled values when a fluorescent lamp is used: because of the higher flickering, the sensor readings are characterized by larger variability with respect to the values shown in Fig. 3(a). In some cases, even when the light is turned on, the acquired values are lower than T.

As shown in Fig. 3(c), the simple adaptlet described above successfully adapted the application behavior to the operating environment.

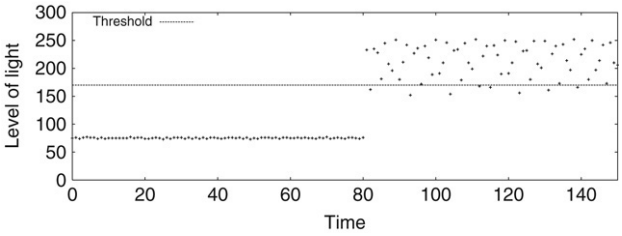
4.2. Filtering noise

Hardware malfunctions and local conditions of the physical environment can be a source of noise for a sensor network application. Sometimes, the noise affecting a single node can be attenuated through the cooperation of neighbouring nodes, which can send information about the surrounding area. In such cases, VAL can return to the application a value that synthesizes the knowledge of a cluster of nodes. For example, a simple spatial aggregation can be computed through the mean value of the ADC readings of neighbouring nodes. The following adaptlet collects the ADC readings from up to 3 neighbours:

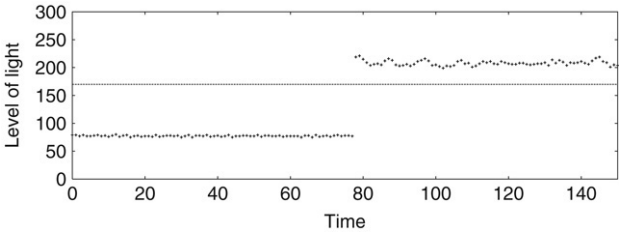
```
0:PUSH 3        (push the maximum number of ADC values that must be
                collected from neighbours)
1:ADCREQ        (request the ADC value to the VCLs of neighbours and
                wait for a given number of readings, as specified by
                the operand on the stack. A timeout is automatically
                set. An integer specifying the number of actual
                sampled readings is placed on the stack top)
2:AVG           (compute the average value)
3:RETURN        (program end)
```



(a) Incandescence lamps.



(b) Fluorescent lamps (before using the adaptation layer).



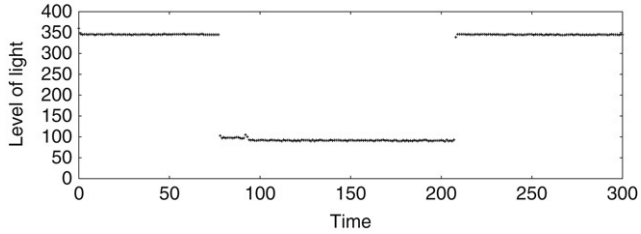
(c) Fluorescent lamps (when the adaptation layer is used).

Fig. 3. Removing flickering caused by fluorescent lamps.

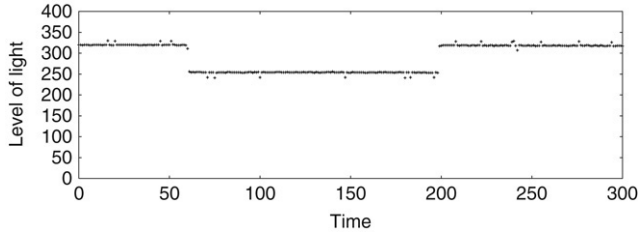
Fig. 4 reports the results of a simple experimental evaluation of noise filtering through spatial aggregation. The experiment was carried out in two rounds. Each round involved four nodes sensing the intensity of light. During each round, one node out of four was exposed to a variation of the level of light. The level of light and its variation were controlled in order to replicate the same conditions for both rounds. During the first round, node 1 was exposed to the variation of light, as shown in Fig. 4(a). During the second round, node 1 was exposed to an identical variation, but this time the node was running the adaptlet described above. As shown in Fig. 4(b), in the latter case the same environmental change led to a smaller fluctuation. This result was due to the compensatory effect of the other three nodes, that were not affected by the variation of the light level.

4.3. Threshold tuning for event detection

A key issue when writing an application for sensor nodes concerns how to set the right threshold for event detection. In fact, it is likely that the phenomenon changes during the



(a) First round: Level of light of node 1.



(b) Second round: Level of light of node 1 running the spatial aggregator.

Fig. 4. Filtering noise through spatial aggregation.

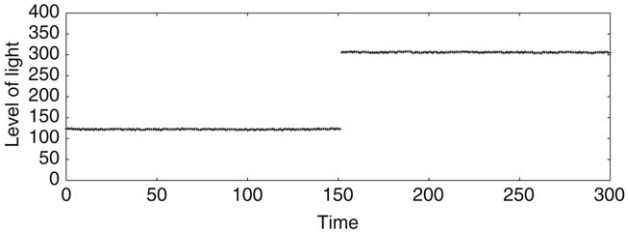


Fig. 5. Threshold tuning.

network lifetime, or that sensors need to be re-calibrated because of deterioration of sensing hardware or other causes (e.g. dust).

In such cases, VAL can be programmed to change the value returned to the application, e.g. by adding/subtracting a given amount from the sampled value in order to correct the effects of deterioration. For example, let us suppose that a node needs a re-calibration of the ADC reading by adding a specific offset (200 in this case). The adaptlet sent from the base station could be as follows (the result of its execution on a real node is shown in Fig. 5):

```
0:PUSH 200      (place the correcting value on the stack)
1:PUSH 10       (push the sampling period)
2:PUSH 1        (push the number of samples)
3:ADCGET        (get the ADC value)
4:POP           (remove the number of samples)
```

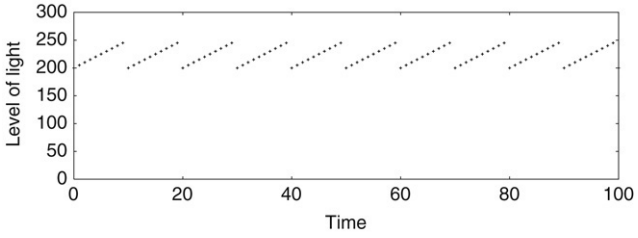


Fig. 6. Emulation of a sawtooth behavior.

```
5:ADD           (add the correcting value)
6:RETURN        (program end)
```

4.4. Emulation of environments and topologies

The first phase of the development of an application for sensor networks is usually carried out on a PC through the use of a simulator that simplifies the burden of compiling, executing and correcting software modules. For example, TOSSim [34] is a simulator that provides an environment for the execution of TinyOS applications. In order to explore all the functionalities of the software under test, programmers specify the values that must be gathered by the nodes. After extensive simulation, the development continues on real hardware, where the application is exposed to operating conditions that are similar to those of the final deployment. As a matter of fact, such phase is much more troublesome and time-consuming.

Let us consider a node that is supposed to react in some way when the sensed value of light becomes greater than a given threshold. To test if the application behaves correctly, the developer has to physically access the specific node and expose it to a source of light. Similarly, if we consider an application that uses a given multi-hop protocol, nodes must be properly placed so that they cannot directly communicate with each other.

The adaptation layer enables developers to test an application on real hardware without actually deploying the nodes: the application under test can be exposed to conditions defined by the programmer in order to emulate the operating environment and/or the network topology. For example, the following adaptlet generates artificial ADC values according to a sawtooth pattern. Values range from 200 to 250, with a step equal to 5. The results obtained on real hardware are shown in Fig. 6.

```
0:LOAD 0      (load the value of the counter from register number 0)
1:PUSH 5      (push the increment step)
2:ADD         (increment the value of the counter)
3:MOD 50      (compute modulo 50)
4:STORE 0     (store the value into register number 0)
5:PUSH 200    (push the minimum value of the signal)
6:ADD         (add the minimum to the value)
7:RETURN      (program end)
```

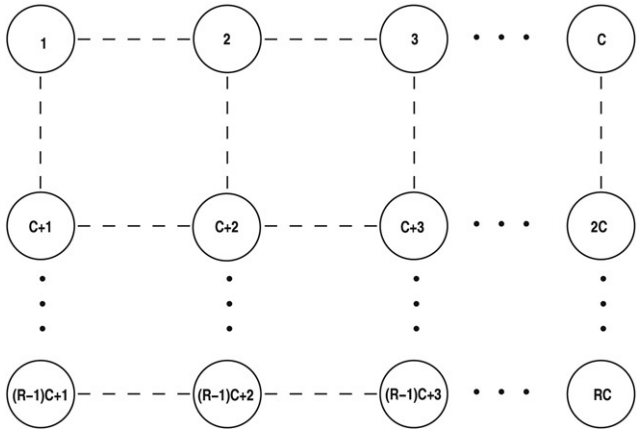


Fig. 7. Emulation of a grid topology (R rows and C columns).

Another adaptlet can be associated to the *receive(. . .)* event to test the multi-hop features of the application when all nodes are in the same radio range. The adaptlet selectively drops packets in order to emulate a grid-like topology similar to the one shown in Fig. 7. Messages sent by nodes that are direct neighbours in the grid are propagated to the application, while all other messages are dropped. Messages that shall be returned to the application are selected on the basis of the sender's address (*senderID*). The following code arranges the nodes in a grid with R rows and C columns:

```
public class Emu {
    static final int R = 4, C = 4;
    static final int Z = ... // offset of the byte that
                             // contains the sender id
    static int LOCAL_ADDRESS;

    static byte[] a0;
    static int r0;//up
    static int r1;//down
    static int r2;//left
    static int r3;//right

    static public void receive(byte[] p){}

    static void init(){
        r0 = r1 = r2 = r3 = 1;
        if(LOCAL_ADDRESS <= C) r0 = 0;
        if(LOCAL_ADDRESS > C*(R-1)) r1 = 0;
        if(LOCAL_ADDRESS % C == 1) r2 = 0;
        if(LOCAL_ADDRESS % C == 0) r3 = 0;
    }
```

```
    static void execute(){
        int senderID = a0[Z];
        //If the packet comes from a direct neighbour
        //and the local node is not located on the
        //border of the grid, the packet is delivered
        //to the application
        if(senderID == LOCAL_ADDRESS + 1 && r3==1 ||
           senderID == LOCAL_ADDRESS - 1 && r2==1 ||
           senderID == LOCAL_ADDRESS + C && r1==1 ||
           senderID == LOCAL_ADDRESS - C && r0==1)
            receive(a0);
    }
}
```

4.5. Aggregation and compression of data

Typically, monitoring applications operate cyclically: they get a value from the sensor board, then they send out a packet that contains the acquired value. While this behavior minimizes latency, it is not optimal in terms of energy consumption. In order to reduce the energy spent for wireless communication, such applications should collect a number of ADC values, and then forward the collected values to the sink by using a single data packet.

The following adaptlet extracts the ADC value contained in the payload and stores it in a buffer: when the buffer is full, a packet containing the set of buffered values is transmitted to the base station. The code shown below is based on these assumptions: the packet header is 10 B long, the maximum length of the payload is 20 B, the first byte of the header specifies the actual length of the payload, and ADC values are represented by a single byte.

```
class Aggr {
    static int r0; //used as a counter
    static void clone(byte[] dst, byte[] src){}
    public static void execute(){
        if(r0==0){
            // the first time the adaptlet is executed
            // the packet issued by the application is
            // duplicated (to copy also the header
            // of the packet)
            clone(dst,src);
        }
        // src[10] is the beginning of the payload: the
        // first 10 bytes are used for the packet header
        // thus src[10] contains the ADC value
        dst[10 + r0] = src[10];
        // the payload of a single packet is able to hold 20 values
        if(r0 == 20){
```

```
            // the counter is reset for the next aggregation
            r0 = 0;
            // dst[0] contains the length of the data payload
            // it is changed to reflect the actual length of the
            // packet that contains the aggregated values
            dst[0] = 20;
            // sends the packet
            send(dst);
        }
        else{
            r0++;
        }
    }
}
```

The technique above becomes useless when the application itself already fills a buffer with ADC values before sending the packet. In this case, if the phenomenon changes slowly with respect to the sampling period, the data payload could be compressed in order to reduce the amount of transmitted data. To this purpose, we carried out an experiment using an adaptlet that implements a very simple lossless compression algorithm, the Run-Length Encoding (RLE) algorithm. The experiment involved a single node sampling the temperature of a room every 60 s. After having collected 20 values, the node sends the data to the base station through a single packet. We installed the adaptlet implementing the RLE algorithm on the node, and we observed a significant compression rate on the data payload (about 7.7 times) and a good compression rate on the amount of bytes transmitted (approximately 2.4 times, since the packet header is not compressed).

5. Execution and reprogramming cost

We experimentally evaluated the overhead introduced by the adaptation layer by measuring the execution time of bytecode instructions on real nodes (Tmotes), and the reprogramming cost of a single node.

5.1. Execution cost

Table 1 summarizes the execution time of a selection of bytecode instructions, grouped according to their categories. To better explain how their overhead affects the overall execution time, we need to describe how the execution of the instructions of a simple adaptlet is scheduled with respect to the application code. Let us suppose that the application operates according to the following code:

```
void task f(){
   ...
   instruction1;
```

Table 1
Execution time of a selection of bytecode instructions

PUSH, POP	61 μs
LOAD, STORE	61 μs
IF ⟨cond⟩, RETURN	61 μs
ADD	62 μs
AVG (4 values)	277μs
AVG (8 values)	369 μs
ADCGET (light)	431 μs
SEND	3890 μs
CLONE	62 μs

```
   ADC.getData();
   instruction2;
   instruction3;
   ...
}

async event result_t ADC.dataReady(uint16_t this_data){
  ...
}
...
```

and let us also suppose that the application is executing task *f()*, and that the adaptlet presented in Section 4.3 is installed in the acquisition library.

During execution, application instructions and adaptlet instructions are interleaved as depicted in Fig. 8(a). After *instruction*1, the application starts a reading from the sensing board by calling the *getData()* command. This call is intercepted by the adaptation layer and the associated adaptlet is scheduled for execution. Instructions that follow the *getData()* invocation, as *instruction*2 and *instruction*3 in our case, are executed until the end of the current task is reached.

When the `ADCGET` instruction is scheduled, the actual reading from the sensing board starts, and the adaptlet is paused until the completion of data acquisition. According to the split phase paradigm, the micro-controller becomes idle, with both the application and the adaptlet waiting for the completion of the sensing activity. During this phase, CPU cycles can be exploited to carry out possible background activity of the application. When the sensing hardware returns a value, the execution of the `ADCGET` instruction is resumed, and the remaining instructions of the adaptlet are executed. Finally, the `RETURN` instruction of the adaptlet signals the *dataReady(...)* event to the application, providing the acquired value properly modified.

Fig. 8(b) shows the execution of the same application when the adaptation layer is not used. Also in this case, after the execution of *instruction*2 and *instruction*3, the micro-controller becomes idle, and remains in this state until the end of the sensing process.

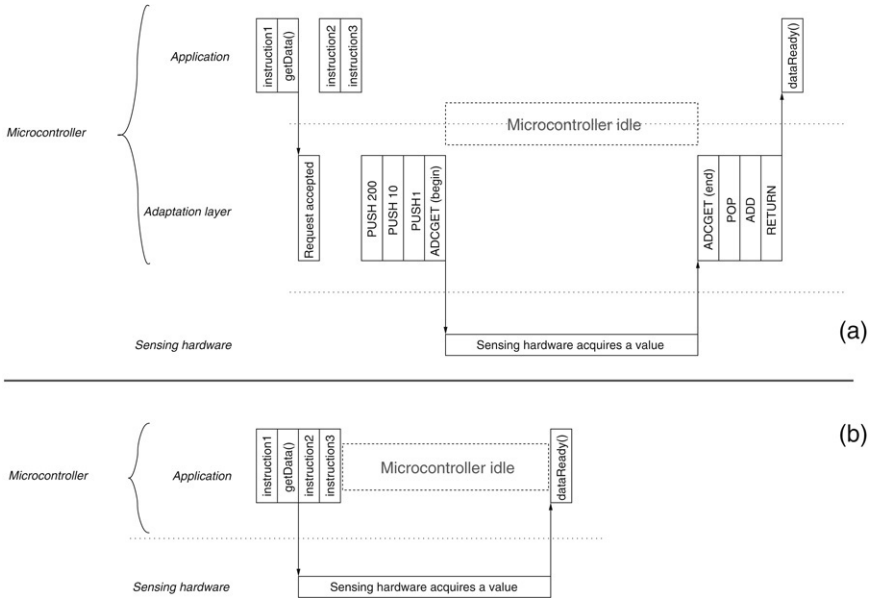


Fig. 8. Execution of application and adaptlet code.

Table 2
Reprogramming by using the adaptation layer and full image replacement

Application adaptation layer		Full image replacement	
Adaptlet size	10 B	Image size	24 066 B
Reprogramming time	213 μs	Reprogramming time	∼41 s
Packets received by the node	1	Packets received by the node	∼1200
Packets sent by the node	1	Packets sent by the node	∼100

In the example above, the total amount of time spent for the execution of the adaptlet code is approximately 800 μs, while the time needed by the sensing hardware to acquire an ADC value is 17.4 ms. Thus, the overhead introduced by the adaptation layer is negligible.

5.2. Reprogramming cost

We evaluated the costs needed to reconfigure an application both when using the adaptation layer and when using full image replacement. The considered scenario involves a single hop network where an application periodically samples the ADC and sends out a packet if the value of the ADC exceeds a fixed threshold. The goal is to modify the application behavior by changing the threshold. Table 2 reports the results of the experiment.

It can be noticed that the adaptation layer requires only a single packet to transfer the adaptlet to the sensor node, while the replacement of the entire image requires more than one thousand packets.

Other techniques, such as the one described in [14], can reduce the cost of full image replacement by a factor of ten (in a scenario similar to the one we described above). Hence,

also in this case the use of the adaptation layer still provides a significant performance improvement.

As a final remark, we are aware that the comparison presented here is not completely fair, since full image replacement allows developers to entirely change the application, while the modifications allowed by the adaptation layer are limited to a set of services. Nevertheless, the comparison highlights the benefits of our approach when a trade-off between costs and adaptability is of importance.

6. Related work

Hadim and Mohamed in [35], propose a classification of middleware approaches for WSNs. A first macroscopic taxonomy can be made by grouping solutions into two broad classes: systems that provide programming abstraction and systems that provide programming support. Approaches that belong to the first category increase the level of abstraction associated with nodes, data, and the network itself. In some systems, such as Kairos [36], the network is considered as a whole, and its global behavior can be defined through high level specification (code to be executed on single nodes is automatically generated). Approaches that belong to the second category do not change the programming model, rather they try to ease the production of code by providing run-time mechanisms such as code distribution, data aggregation and manipulation, management of messages. Our proposed technique falls into this category, as it provides mechanisms to increase the flexibility of applications without changing the programming model. In the following we compare our solution with respect to similar systems (i.e. systems that allow reconfiguration and/or re-programmability of WSNs).

A first technique to reprogram the network consists in remotely changing the binary image installed on sensor nodes. XNP [12] and Deluge [13] are two systems that enable network reprogramming of sensor nodes based on TinyOS. They provide mechanisms to send the binary image over the wireless channel. The image is saved in flash memory and then loaded as new image. These approaches ensure maximum flexibility, since they enable a complete change of the application code, but they are expensive in terms of energy requirements. Application code is split into a large number of packets that must be delivered to all nodes in the network, requiring numerous transmissions.

Techniques for incremental network reprogramming for TinyOS are proposed in [14, 15]. Incremental network programming optimizes the transmission of the binary image since only the difference between the old and the new image is actually transmitted over the wireless channel. These approaches speed up the update process, but still suffer from high energy consumption since the binary image has to be written entirely in flash memory.

Alternatively, applications can be designed to operate on different and more flexible operating systems, such as SOS [18]. SOS consists of a common kernel and dynamic application modules. Application modules can be substituted during run-time by injecting the new version of the modules into the network. This solution ensures modular adaptation of executing applications.

Other approaches are based on the active sensor concept where nodes are able to react to events in a customizable fashion. Reaction policies can be disseminated through some

```
configuration Oscilloscope{}
implementation{
  components Main, OscilloscopeM,
             ... , LightSensorC;

  Main.StdControl -> OscilloscopeP;
  ...
  OscilloscopeM.SensorControl -> LightSensorC;
  OscilloscopeM.SensorADC     -> LightSensorC;
}
```

(a) Original.

```
configuration Oscilloscope{}
implementation{
  components Main, OscilloscopeM,
             ... , LightSensorC, AAL;

  Main.StdControl -> OscilloscopeP;
  ...
  OscilloscopeM.SensorControl -> AAL;
  OscilloscopeM.SensorADC     -> AAL;
  AAL.SensorControl -> LightSensorC;
  AAL.SensorADC     -> LightSensorC;
}
```

(b) Modified.

Fig. 9. OscilloscopeRF.nc (some details are not shown).

form of mobile code. Some solutions adopt tiny virtual machines specifically designed for sensor networks (for example Maté [19] and application specific virtual machines [8]). Others, like SensorWare [37], strive to change the run-time abstraction in an application-specific fashion by dynamically installing new services.

Solutions based on virtual machines and modular operating systems offer a high degree of flexibility. Nevertheless, adopting such solutions is usually impractical with already existing applications (mostly written for the nesC/TinyOS platform), as they require to re-write the application code according to the interfaces offered by the new operating system or the abstractions of the specific virtual machine.

7. Conclusions

Configuring and tuning applications for sensor nodes usually implies changing just a small fraction of their code. Nevertheless, providing an efficient paradigm to change specific sections of code is still a challenge.

We presented a solution based on the idea of placing a programmable computation layer at the interface between the application logic and the operating system. If compared to other solutions, the adaptation layer approach can be less flexible, since it does not allow the developer to change the entire code of a running application, but only its communication scheme or the way it perceives the surrounding environment. On the other hand, differently from other systems, the adaptation layer can be adopted on already available applications without the need of re-writing the application logic, and limiting the amount of traffic generated during reconfiguration. Therefore, it can be a practical and effective solution to *configure* and *tune* sensor network applications.

The experiments confirmed the capabilities of the adaptation layer. The proposed system is not invasive and can be used with minimal changes to existing applications. Some of the experiments were carried out by using the *OscilloscopeRF* application, included in the TinyOS distribution. Such application consists of two files: *OscilloscopeM.nc*, which contains more than 100 lines of code, implements the application logic; *Oscilloscope.nc*, which contains about 20 lines of code, is the main configuration file that wires the application logic to the operating system, sensor–driver and networking components. In order to incorporate the adaptation layer, we had to modify the configuration file and, as shown in Fig. 9, the extent of required modifications was limited to a few lines.

Appendix. The instruction set

Mnemonic	Description
`PUSH v`	Places the integer `v` on top of the operand stack.
`POP`	Removes the topmost element of the operand stack.
`STORE r`	Stores the topmost element of operand stack into register `r`. The topmost element is removed from the operand stack.
`LOAD r`	Loads the content of register `r` on top of the operand stack.
`DUP`	Duplicates the topmost value of the operand stack.
`GET r`	Places the address of register `r` on top of the operand stack.
α`ALOAD`	Loads a value of type $\alpha \in \{\text{int8\_}t, \text{int16\_}t\}$ from a buffer to the operand stack. The buffer is interpreted as array of αs. The instruction takes the operands from the stack: the address of the buffer is the next-to-top element of the operand stack, the index in the buffer is the topmost element of the operand stack. The operands are removed from the stack, and the value of type α is placed on the stack top.
α`ASTORE`	Stores a value of type $\alpha \in \{\text{int8\_}t, \text{int16\_}t\}$ from the operand stack to a buffer. The buffer is interpreted as array of αs. The instruction takes the operands from the stack: the address of the buffer is the next-to-next-to-top element of the operand stack, the index in the buffer is the next-to-top element of the operand stack, the value of type α is the topmost element of the operand stack. The operands are removed from the stack and the buffer is updated.
`CLONE`	Copies a buffer. The address of the buffer to be copied is the next-to-top element on the operand stack. The destination buffer is the topmost element on the operand stack.
`GOTO` *l*	Jumps to instruction labelled with number *l*.
`IF_CMP`*cond l*	Compares the topmost value of the operand stack with the next-to-top value. The possible conditions *cond* are LE (less or equal), GE (greater or equal), NE (not equal), EQ (equal), LT (lower), GT (greater). If the condition succeeds then jumps to instruction labelled with number *l*. Otherwise, the instruction jumps to the instruction following the `IF_`*cond* instruction. The operands are removed from the operand stack.
`IF`*cond l*	Compares the topmost value of the operand stack with zero. Possible conditions and behavior is like instruction `IF_CMP`*cond l*.
`RETURN`	Terminates the execution of the instruction sequence.
`ADD/SUB/MUL/DIV`	Arithmetic instructions for integers. The left operand is the next-to-top element on the operand stack, the right operand is the topmost element on the operand stack. The operands are removed from the operand stack and the result is placed on top of the operand stack.

Mnemonic	Description
AVG/MAX/MIN	Computes the average/maximum/minimum value among N values of the operand stack. The number N is specified by the topmost element of the operand stack. The $N + 1$ topmost values are removed from the operand stack and the result is placed onto the operand stack.
GAUSS	Computes a random Gaussian value. The mean value and the standard deviation are specified on the operand stack. The two topmost values of the operand stack are removed and the Gaussian value is placed onto the operand stack.
MOD v	Computes the modulo v. The topmost element of the operand stack is replaced with the result of the modulo operation.
ADCGET	Samples the local sensor N times with period T (expressed in 100's of milliseconds). The number N of samples that must be collected and the sampling period T are specified by the two topmost values of the operand stack. The 2 elements are removed and the sampled values and the number N are placed on the operand stack.
ADCREQ	Requests a sensor reading from the neighbours. The number N on the top of the operand stack specifies the maximum number of replies that must be collected. A timeout proportional to the number of required data is automatically set. When the timer expires, then $M \leq N$ values received from the neighbours are placed on the operand stack, together with the value of the local sensor, and the number $M + 1$.
SEND	Sends out a packet. The address of the buffer containing the packet is specified on top of the operand stack. The fields of the packets are left unchanged and are assumed to be correct. The instruction completes by returning the success code on the operand stack when the packet has been actually sent through the network library.
RECEIVE	Signals the reception of a packet to the application. The address of the buffer containing the received packet is specified on top o f the operand stack. The fields of the packets are left unchanged and are assumed to be correct.

References

[1] J. Hill, M. Horton, R. Kling, L. Krishnamurthy, The platforms enabling wireless sensor networks, Communications of the ACM 47 (6) (2004) 41–46.

[2] I. Akyldiz, W. Su, Y. Sankarasubramaniam, E. Cayirci, Wireless sensor network: A survey, Computer Networks 38 (2002) 393–422.

[3] A. Arora, P. Dutta, S. Bapat, V. Kulathumani, H. Zhang, V. Naik, V. Mittal, H. Cao, M. Demirbas, M. Gouda, Y. Choi, T. Herman, S. Kulkarni, U. Arumugam, M. Nesterenko, A. Vora, M. Miyashita, A line in the sand: A wireless sensor network for target detection, classification, and tracking, Computer Networks 46 (2004) 605–634.

[4] P. Juang, H. Oki, Y. Wang, M. Martonosi, L.S. Peh, D. Rubenstein, Energy-efficient computing for wildlife tracking: Design tradeoffs and early experiences with zebranet, in: Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS-X, ACM Press, New York, NY, USA, 2002, pp. 96–107.

[5] J. Burrell, T. Brooke, R. Beckwith, Vineyard computing: Sensor networks in agricultural production, Pervasive Computing 3 (1) (2004) 38–45.

[6] N. Xu, S. Rangwala, K.K. Chintalapudi, D. Ganesan, A. Broad, R. Govindan, D. Estrin, A wireless sensor network for structural monitoring, in: Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems, SenSys, ACM Press, New York, NY, USA, 2004, pp. 13–24.

[7] A. Lédeczi, A. Nádas, P. Völgyesi, G. Balogh, B. Kusy, J. Sallai, G. Pap, S. Dóra, K. Molnár, M. Maróti, G. Simon, Countersniper system for urban warfare, ACM Transactions on Sensor Networks 1 (2) (2005) 153–177.

[8] P. Levis, D. Gay, D. Culler, Active sensor networks, in: Proceedings of the 2nd USENIX/ACM Symposium on Network Systems Design and Implementation, NSDI, 2005, pp. 343–356.

[9] D. Zhang, H. Ma, L. Liu, D. Tao, An approach to reliable scripts dissemination in wireless sensor networks, in: Proceedings of the Second International Conference on Embedded Software and Systems, ICESS, IEEE Computer Society, Washington, DC, USA, 2005, pp. 438–446.

[10] T. He, S. Krishnamurthy, L. Luo, T. Yan, L. Gu, R. Stoleru, G. Zhou, Q. Cao, P. Vicaire, J.A. Stankovic, T.F. Abdelzaher, J. Hui, B. Krogh, Vigilnet: An integrated sensor network system for energy-efficient surveillance, ACM Transactions on Sensor Networks 2 (1) (2006) 1–38.

[11] C. Han, R. Kumar, R. Shea, M. Srivastava, Sensor network software update management: A survey, International Journal of Network Management 15 (1099–1190) (2005) 283–294.

[12] Crossbow Technology Inc., Mote In-Network Programming User Reference, 2003.

[13] J.W. Hui, D. Culler, The dynamic behavior of a data dissemination protocol for network programming at scale, in: Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems, ACM Press, 2004, pp. 81–94.

[14] J. Jeong, D. Culler, Incremental network programming for wireless sensors, in: First Annual IEEE Conference on Sensor and Ad Hoc Communications and Networks, SECON, IEEE Press, 2004, pp. 25–33.

[15] N. Reijers, K. Langendoen, Efficient code distribution in wireless sensor networks, in: Proceedings of the 2nd ACM International Conference on Wireless Sensor Networks and Applications, WSNA, ACM Press, New York, NY, USA, 2003, pp. 60–67.

[16] P.K. Dutta, D.E. Culler, System software techniques for low-power operation in wireless sensor networks, in: Proceedings of the IEEE/ACM International Conference on Computer-Aided Design, ICCAD, IEEE Computer Society, Washington, DC, USA, 2005, pp. 925–932.

[17] J. Koshy, R. Pandey, Remote incremental linking for energy-efficient reprogramming of sensor networks, in: Proceedings of the Second European Workshop on Wireless Sensor Networks, 2005, pp. 354–365.

[18] C. Han, R. Kumar, R. Shea, E. Kohler, M. Srivastava, A dynamic operating system for sensor nodes, in: Proceedings of the 3rd International Conference on Mobile Systems, Applications, and Services, MobySys, ACM Press, New York, NY, USA, 2005, pp. 163–176.

[19] P. Levis, D. Culler, Maté: A tiny virtual machine for sensor networks, in: Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS-X, ACM Press, New York, NY, USA, 2002, pp. 85–95.

[20] M.U. Arumugam, Infuse: A TDMA based reprogramming service for sensor networks, in: Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems, SenSys, ACM Press, New York, NY, USA, 2004, pp. 281–282.

[21] P. Levis, N. Patel, D. Culler, S. Shenker, Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks, in: Proceedings of the First USENIX/ACM Symposium on Network Systems Design and Implementation, NSDI, USENIX/ACM, 2004, pp. 15–28.

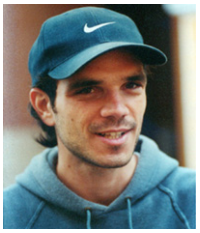
[22] S.S. Kulkarni, L. Wang, MNP: Multihop network reprogramming service for sensor networks, in: Proceedings of the 25th IEEE International Conference on Distributed Computing Systems, ICDCS, IEEE Computer Society, Washington, DC, USA, 2005, pp. 7–16.

[23] T. Stathopoulos, J. Heidemann, D. Estrin, A remote code update mechanism for wireless sensor networks, CENS Technical Report 30, 2003.

[24] P. Corsini, P. Masci, A. Vecchio, Configuration and tuning of sensor network applications through virtual sensors, in: Proceedings of the Fourth IEEE International Conference on Pervasive Computing and Communications Workshops, IEEE Press, 2006, pp. 316–320.
[25] P. Corsini, P. Masci, A. Vecchio, Virtus: A configurable layer for post-deployment adaptation of sensor networks, in: Proceedings of the International Conference on Wireless and Mobile Communications, IEEE Computer Society Press, 2006, p. n.a.
[26] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, K. Pister, System architecture directions for networked sensors, SIGPLAN Notices 35 (11) (2000) 93–104.
[27] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, D. Culler, The nesC language: A holistic approach to networked embedded systems, SIGPLAN Notices 38 (5) (2003) 1–11.
[28] T. von Eicken, D.E. Culler, S.C. Goldstein, K.E. Schauser, Active messages: A mechanism for integrated communication and computation, in: Proceedings of the 19th annual International Symposium on Computer Architecture, ISCA, ACM Press, New York, NY, USA, 1992, pp. 256–266.
[29] D. Gay, P. Levis, D. Culler, Software design patterns for TinyOS, in: Proceedings of ACM LCTES.
[30] I. Stojmenovic, J. Wu, Broadcasting and activity-scheduling in ad hoc networks, in: S.G.S. Basagni, M. Conti, I. Stojmenovic (Eds.), Mobile Ad Hoc Networking, IEEE/Wiley, 2004, pp. 205–229.
[31] I. Stojmenovic, Geocasting with guaranteed delivery in sensor networks, IEEE Wireless Communications (2004) 29–37.
[32] Apache Foundation, ByteCode Engineering Library (BCEL) User Manual, 2002. http://jakarta.apache.org/bcel/index.html.
[33] Moteiv Inc. http://www.moteiv.com.
[34] P. Levis, N. Lee, M. Welsh, D. Culler, TOSSim: Accurate and scalable simulation of entire TinyOS applications, in: Proceedings of the First International Conference on Embedded Networked Sensor Systems, SenSys, ACM Press, New York, NY, USA, 2003, pp. 126–137.
[35] S. Hadim, N. Mohamed, Middleware challenges and approaches for wireless sensor networks, IEEE Distributed Systems Online 7 (3).
[36] R. Gummadi, O. Gnawali, R. Govindan, Macro-programming wireless sensor networks using kairos, in: Proceedings of the International Conference on Distributed Computing in Sensor Systems, 2005, pp. 126–140.
[37] A. Boulis, C. Han, R. Shea, M. Srivastava, SensorWare: Programming sensor networks beyond code update and querying, Pervasive and Mobile Computing 3 (4) (2007) 386–412.

M. Avvenuti is an Associate Professor with the Department of Information Engineering of the University of Pisa. He received his Ph.D. in Information Engineering from the University of Pisa in 1993. In 1992 and 1996 he was a visiting scholar at the Berkeley's International Computer Science Institute (ICSI). His current research interests are in the areas of distributed object computing, mobile computing and ad hoc networks. He was involved in several projects on middleware aspects of mobile computing, funded by the European Community, the Italian Ministry of Education and Research and by the Italian National Council of Research (CNR).

P. Corsini is Full Professor with the Dipartimento di Ingegneria dell'Informazione of the University of Pisa. He has been working on fuzzy systems, clustering algorithms, and distributed systems. Currently his interests are in the area of sensor networks.



P. Masci received the Laurea degree in Computer Engineering from the University of Pisa in 2003. Currently, he is a Ph.D. student at the University of Pisa, Department of Information Engineering. He has been working on Java Cards, focusing on bytecode verification and secure information flow. During the last few years he has started to work on wireless sensor networks, focusing on abstraction layers for post-deployment configuration of sensor nodes.

A. Vecchio received his Ph.D. in Information Engineering from the University of Pisa in 2003. Currently, he works as a researcher at the Department of Information Engineering, where his activity is related to pervasive and mobile computing. His research interests also include distributed object systems and network emulation.